



Usage and Customization Guide

OpenL Tablets Rule Service
Release 5.20



Document number: TP_OpenL_WebServices_UCG_2.4_LSh

Revised: 11-28-2018



OpenL Tablets Documentation is licensed under a [Creative Commons Attribution 3.0 United States License](https://creativecommons.org/licenses/by/3.0/).

Table of Contents

1	Preface	5
1.1	Audience.....	5
1.2	How This Guide Is Organized.....	5
1.3	Related Information.....	5
1.4	Typographic Conventions.....	6
2	Introduction	7
3	OpenL Tablets Rule Service	9
3.1	Adding Dependencies into the Project.....	9
3.2	Configuring Spring Integration for OpenL Tablets Rule Service.....	9
	Adding a Bean Configuration File to the Spring Context Definition.....	9
	Simple Java Frontend Implementation.....	9
3.3	Customizing and Configuring OpenL Tablets Rule Service.....	10
4	OpenL Tablets Web Services Configuration	11
4.1	OpenL Tablets Web Services Default Configuration.....	11
4.2	OpenL Tablets Web Services Default Configuration Files.....	11
4.3	Service Manager.....	12
4.4	Configuration Points.....	12
	Configuring a Data Source.....	13
	Configuring System Settings.....	19
	Configuring a Number of Threads to Rules Compilation.....	19
	Logging Requests to OpenL Tablets Web Services and Their Responnds.....	19
	Configuring REST Services Settings.....	21
	Configuring RMI Services Settings.....	21
	Configuring Aegis Databinding.....	21
	Configuring the Instantiation Strategy.....	21
	Configuring the Deployment Filter.....	22
5	OpenL Tablets Web Services Customization	23
5.1	OpenL Tablets Web Services Customization Algorithm.....	23
5.2	Service Configurer.....	24
	Understanding Service Configurer.....	24
	Deployment Configuration File Used by Service Configurer.....	25
	Service Description.....	25
5.3	Multimodule with Customized Dispatching.....	26
5.4	Dynamic Interface Support.....	27
5.5	Interface Customization through Annotations.....	28
	Interceptors for Service Methods.....	28
	Annotation Customization for Dynamic Interfaces.....	29
	Understandings Groups in Annotations.....	30
5.6	JAR File Data Source.....	31
5.7	Data Source Listeners.....	31
5.8	Service Publishing Listeners.....	31
5.9	Variations.....	32
	Variation Algorithm.....	33

Predefined Variations	33
Variations Factory	34
Variations as Rules	35
Example.....	35
6 Appendix A: Tips and Tricks	36
6.1 Using OpenL Tablets Web Services from Java Code.....	36
6.2 Using OpenL Tablets REST Services from Java Code	37

1 Preface

OpenL Tablets is a Business Rules Management System (BRMS) based on the tables presented in Excel and Word documents. Using unique concepts, OpenL Tablets facilitates treating business documents containing business logic specifications as executable source code.

OpenL Tablets provides a set of tools addressing BRMS related capabilities including *OpenL Tablets Web Services application* designed for integration of business rules into different customers' applications.

The goal of this document is to explain how to configure OpenL Tablets Rule Service, that is, configure OpenL Tablets Web Services application or integrate the OpenL Tablets Rule Service module into the existing application, for different working environments and how to customize the services to meet particular customer requirements.

The following topics are included in this chapter:

- [Audience](#)
- [How This Guide Is Organized](#)
- [Related Information](#)
- [Typographic Conventions](#)

1.1 Audience

This guide is targeted at rule developers who integrate the OpenL Tablets Rule Service module and set up, configure, and customize OpenL Tablets Web Services to facilitate the needs of customer rules management applications.

Basic knowledge of Java, Apache Tomcat, Ant, Maven, and Excel is required to use this guide effectively.

1.2 How This Guide Is Organized

Information on how to use this guide	
Section	Description
Introduction	Provides overall information about OpenL Tablets Web Services application.
OpenL Tablets Rule Service	Introduces OpenL Tablets Rule Service functionality.
OpenL Tablets Web Services Configuration	Describes the configuration of OpenL Tablets Web Services for different environments.
OpenL Tablets Web Services Customization	Explains how to customize OpenL Tablets Web Services to meet customers' needs and requirements.
Appendix A: Tips and Tricks	Describes how to use OpenL Tablets Web Services from Java code.

1.3 Related Information

The following table lists sources of information related to contents of this guide:

Related information	
Title	Description
[OpenL Tablets WebStudio User Guide]	Describes OpenL Web Studio, a web application for managing OpenL Tablets projects through web browser.
[OpenL Tablets Reference Guide]	Provides overview of OpenL Tablets technology, as well as its basic concepts and principles.
[OpenL Tablets Installation Guide]	Describes how to install and set up OpenL Tablets software.
http://openl-tablets.org/	OpenL Tablets open source project website.

1.4 Typographic Conventions

The following styles and conventions are used in this guide:

Typographic styles and conventions	
Convention	Description
Bold	<ul style="list-style-type: none"> Represents user interface items such as check boxes, command buttons, dialog boxes, drop-down list values, field names, menu commands, menus, option buttons, perspectives, tabs, tooltip labels, tree elements, views, and windows. Represents keys, such as F9 or CTRL+A. Represents a term the first time it is defined.
<code>Courier</code>	Represents file and directory names, code, system messages, and command-line commands.
Courier Bold	Represents emphasized text in code.
Select File > Save As	Represents a command to perform, such as opening the File menu and selecting Save As .
<i>Italic</i>	<ul style="list-style-type: none"> Represents any information to be entered in a field. Represents documentation titles.
< >	Represents placeholder values to be substituted with user specific values.
Hyperlink	Represents a hyperlink. Clicking a hyperlink displays the information topic or external source.
[name of guide]	Reference to another guide that contains additional information on a specific feature.

2 Introduction

Many OpenL Tablets rule management solutions need to expose business rules as web services. Each solution usually has a unique structure of the rules and implies a unique structure of web services. To meet requirements of a variety of customer project implementations, OpenL Tablets Web Services provides the ability to dynamically create web services for customer rules and offers extensive configuration and customization capabilities.

Overall architecture of OpenL Tablets Web Services frontend is expandable and customizable. All functionality is divided into pieces; each of them is responsible for a small part of functionality and can be replaced by another implementation.

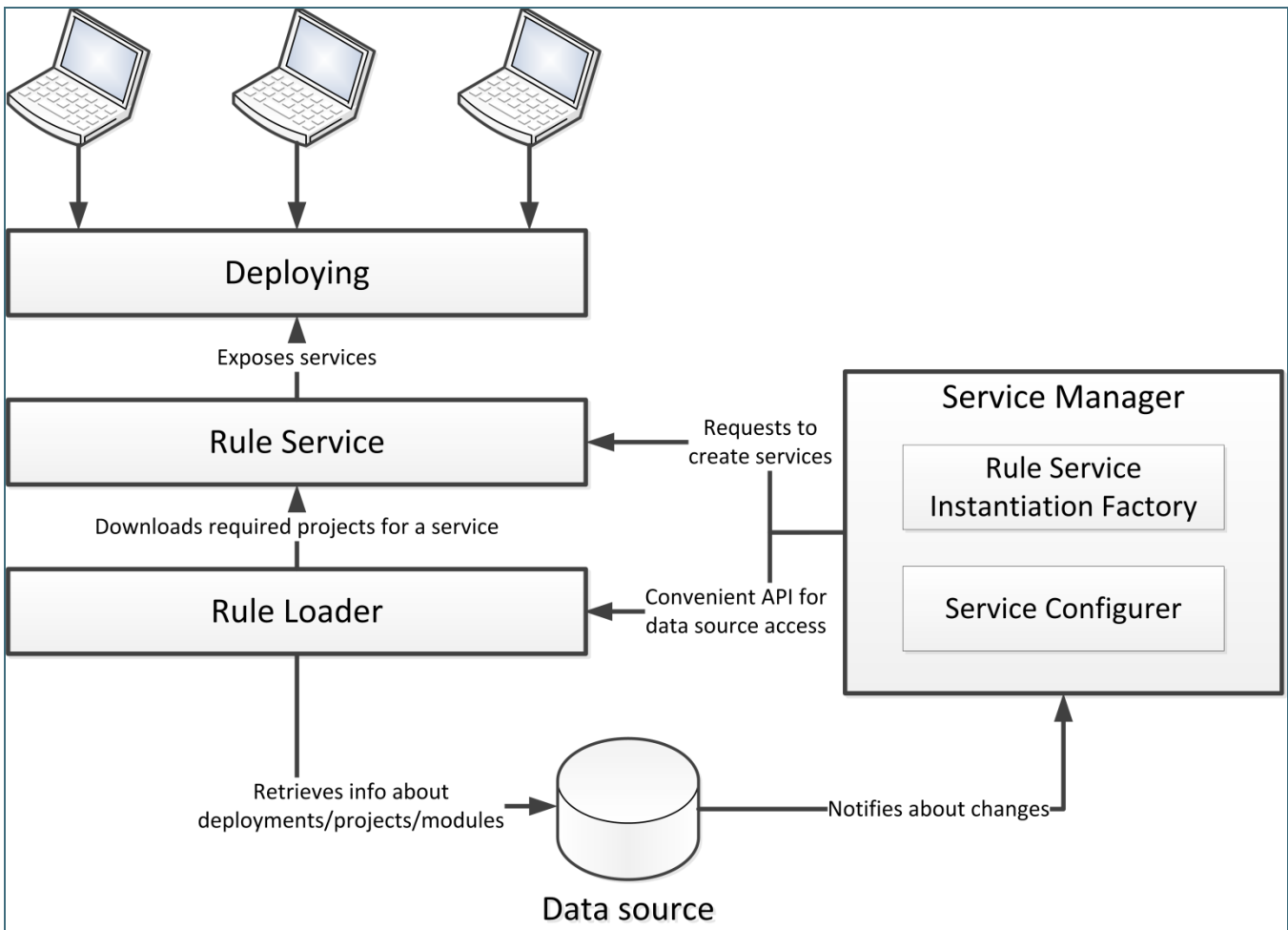


Figure 1: Overall OpenL Tablets Web Services architecture

OpenL Tablets Web Services application provides the following key features and benefits:

- easily integrating customer business rules into various applications running on different platforms
- using different data sources, such as a central OpenL Tablets production repository or file system of a proper structure
- exposing multiple projects and modules as a single web service according to a project logical structure

The subsequent chapters describe how to set up a data source, Service Configurer, and a service exposing method, and how to integrate OpenL Tablets into the existing application.

The OpenL Tablets Web Services application is based on OpenL Tablets Rule Service and supports all features that provided by OpenL Tablets Rule Service module.

The following diagram identifies all components to be configured and customized.

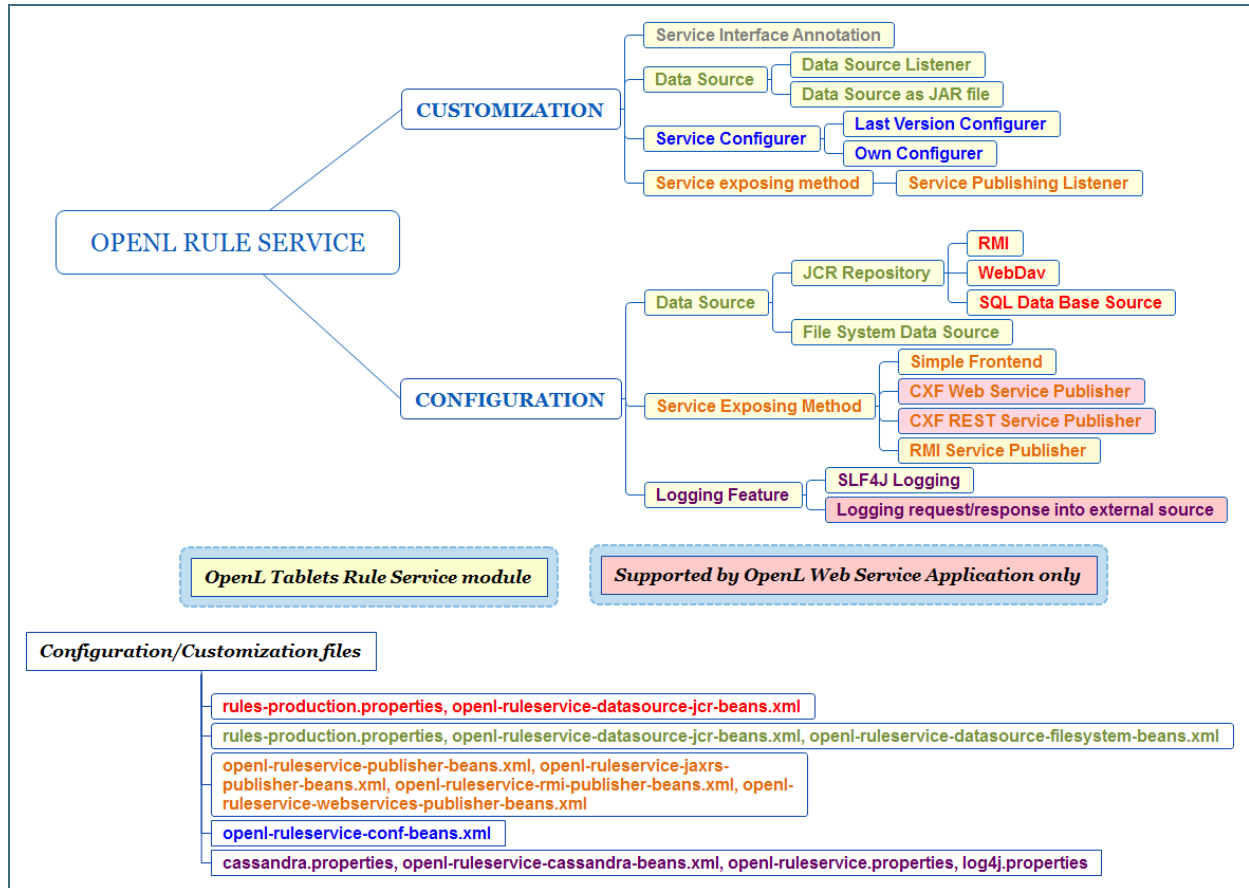


Figure 2: Configurable and customizable components of OpenL Tablets Rule Service

3 OpenL Tablets Rule Service

This section introduces OpenL Tablets Rule Service functionality and includes the following topics:

- [Adding Dependencies into the Project](#)
- [Configuring Spring Integration for OpenL Tablets Rule Service](#)
- [Customizing and Configuring OpenL Tablets Rule Service](#)

3.1 Adding Dependencies into the Project

To use the OpenL Tablets Rule Service within Maven, declare the module dependencies in the project object model (POM) as described in the following example:

```
<dependency>
  <groupId>org.openl.rules</groupId>
  <artifactId> org.openl.rules.ruleservice</artifactId>
  <version>${openl.version}</version>
</dependency>
```

If Apache Maven is not used in the project, it is recommended to download all dependencies via Maven and add all downloaded dependencies into the existing project classpath.

3.2 Configuring Spring Integration for OpenL Tablets Rule Service

This section describes how to configure Spring and OpenL Tablets Rule Service integration and includes the following topics:

- [Adding a Bean Configuration File to the Spring Context Definition](#)
- [Simple Java Frontend Implementation](#)

Adding a Bean Configuration File to the Spring Context Definition

To support the OpenL Tablets Rule Service features, add the `openl-ruleservice-beans.xml` bean configuration file into the application Spring context definition. An example is as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  <import resource="classpath:openl-ruleservice-beans.xml" />
</beans>
```

After adding the OpenL Tablets Rule Service beans, Spring configuration has a simple Java frontend service as a default publisher for all OpenL Tablets services.

Simple Java Frontend Implementation

Spring configuration of the bean used for simple Java frontend implementation is as follows:

```
<!-- Simple front end to access all services. -->
<bean id="frontend" class="org.openl.rules.ruleservice.simple.RulesFrontendImpl"/>
```

```
<!-- Initializes OpenL Engine instances according to OpenL Tablets Web Services configuration
description and calls DeploymentAdmin to expose corresponding web service. -->
<bean id="ruleServicePublisher"
class="org.openl.rules.ruleservice.simple.JavaClassRuleServicePublisher">
    <property name="frontend" ref="frontend"/>
</bean>
```

The `frontend` bean implements the `org.openl.rules.ruleservice.simple.RulesFrontend` interface. This bean is designed to interact with deployed OpenL Tablets services.

Method in <code>org.openl.rules.ruleservice.simple.RulesFrontend</code>	
Inceptor	Description
<code>Object execute(String serviceName, String ruleName, Class<?>[] inputParamsTypes, Object[] params)</code>	Invokes a rule with defined parameter types and parameter values from the deployed OpenL Tablets service.
<code>Object execute(String serviceName, String ruleName, Object... params)</code>	Invokes a rule with defined parameter values from the deployed OpenL service. Parameter types are automatically defined from send parameters.
<code>Object getValue(String serviceName, String fieldName)</code>	Returns field value from the defined OpenL Tablets service.
<code>Collection<String> getServiceNames()</code>	Returns a list of registered OpenL Tablets services.
<code>void registerService(OpenLService service)</code>	Registers the OpenL Tablets service.
<code>void unregisterService(String serviceName)</code>	Unregisters the OpenL Tablets service.
<code><T> T buildServiceProxy(String serviceName, Class<T> proxyInterface)</code>	Builds a proxy for the OpenL Tablets service with a defined interface.
<code>T> T buildServiceProxy(String serviceName, Class<T> proxyInterface, ClassLoader classLoader)</code>	Builds a proxy for the OpenL Tablets service with a defined interface and defined class loader.

To inject the `frontend` bean into the application beans or build a new proxy bean and use it in the application, proceed as described in the following example:

```
<bean id="service1" class="org.openl.rules.ruleservice.simple.OpenLServiceFactoryBean">
    <!-- <property name="rulesFrontend" ref="frontend"/> optional. For custom
implementation of RulesFrontend -->
    <property name="serviceName" value="service1"/>
    <property name="proxyInterface" value="com.myproject.Service1"/>
</bean>
```

In this example, `serviceName` is a deployed OpenL Tablets service name and `proxyInterface` is deployed service interface or an interface similar to the OpenL Tablets service rules.

Note: Proxy beans and proxy objects created by `frontend` bean are automatically updated if the OpenL Tablets service is redeployed into a data source. Nevertheless, these objects are not working while the project is redeployed. To synchronize this process, use Service Publisher listeners described in further sections.

3.3 Customizing and Configuring OpenL Tablets Rule Service

The OpenL Tablets Rule Service module configuration features resemble configuration features for the OpenL Tablets Web Services application. The OpenL Tablets Web Services customization and configuration information is provided in this document and can be applied to OpenL Tablets Rule Service in the same way. For the list of components supported only by OpenL Tablets Web Services application, see diagrams in [Introduction](#).

4 OpenL Tablets Web Services Configuration

OpenL Tablets Web Services architecture allows extending mechanisms of services loading and deployment according to the particular project requirements.

This section describes OpenL Tablets Web Services configuration and includes the following topics:

- [OpenL Tablets Web Services Default Configuration](#)
- [OpenL Tablets Web Services Default Configuration Files](#)
- [Service Manager](#)
- [Configuration Points](#)

4.1 OpenL Tablets Web Services Default Configuration

All OpenL Tablets Web Services configuration is specified in Spring configuration files and `application.properties` files. The `application.properties` file is located inside the application `.war` file (inside `WEB-INF/classes` folder), in a user's directory or in a working directory.

The file located inside `.war` file is generated with default settings. You can use it as a reference of possible settings and redefine needed ones in your configuration file (for example `application.properties` file located in a user's home directory).

The settings can be defined as JVM options for Tomcat launch. In this case, JVM options override settings from `application.properties` file.

By default, OpenL Tablets Web Services application is configured as follows:

1. A data source is configured as `FileSystemDataSource` located in the "`${user.home}/.openl/datasource`" folder.
2. All services are exposed using the CXF framework inside the OpenL Tablets Web Services war file that can be downloaded at <http://openl-tablets.org/downloads>.
3. All calls are processed by CXF servlet.
4. `LastVersionProjectsServiceConfigurer` is used as a default service configurer that takes the last version of each deployment and creates the service for each project using all modules contained in the project.

4.2 OpenL Tablets Web Services Default Configuration Files

If necessary, modify the OpenL Tablets Web Services configuration by overriding the existing configuration files. All overridden beans must be located in the `openl-ruleservice-override-beans.xml` file. The following table lists default OpenL Tablets Web Services configuration files:

Default OpenL Tablets Web Services configuration files	
File	Description
<code>openl-ruleservice-beans.xml</code>	Main configuration file that includes all other configuration files. This file is searched by OpenL Tablets Web Services in the classpath root.
<code>openl-ruleservice-datasource-beans.xml</code>	File storing data source configuration.
<code>openl-ruleservice-loader-beans.xml</code>	File storing loader configuration.

Default OpenL Tablets Web Services configuration files	
File	Description
<code>openl-ruleservice-publisher-beans.xml</code>	File storing common publisher configuration.
<code>openl-ruleservice-webservice-publisher-beans.xml</code>	File storing publisher configuration for web services (SOAP).
<code>openl-ruleservice-jaxrs-publisher-beans.xml</code>	File storing publisher configuration for RESTful services.
<code>openl-ruleservice-rmi-publisher-beans.xml</code>	Contains publisher configuration for RMI services.
<code>openl-ruleservice-conf-beans.xml</code>	File storing Service Configurer.
<code>application.properties</code>	Main file containing properties for OpenL Tablets Web Services configuration.
<code>project-resolver-beans.xml</code>	Configuration for OpenL Tablets project resolving. It stores beans for reading rules from the data source specified in the loader.

For more information on configuration files, see [Configuration Points](#).

4.3 Service Manager

Service Manager is the main component of OpenL Tablets Web Services frontend containing all major parts, such as a loader, a rule service, and Service Configurer. For more information on OpenL Tablets Web Services frontend components, see [\[OpenL Tablets Developer Guide\]](#).

Service Manager stores information about all currently running services and intelligently controls all operations for deploying, undeploying, and redeploying the services. These operations are only performed in the following cases:

- initial deployment at startup of the OpenL Tablets Web Services frontend
- processing after data source update

Service Manager always acts as a data source listener as described in further sections of this chapter.

4.4 Configuration Points

Any part of OpenL Tablets Web Services frontend can be replaced by the user's own implementation. For more information on the system architecture, see [\[OpenL Tablets Developer Guide\]](#).

If the common approach is used, the following components must be configured:

Configuration components	
Component	Description
Data source	Informs the OpenL Tablets system where to retrieve user's rules.
Service exposing method	Defines the way services are exposed, for example, as a web service or a simple Java framework.

The following sections describe how to configure these components:

- [Configuring a Data Source](#)

- [Configuring System Settings](#)
- [Configuring a Number of Threads to Rules Compilation](#)
- [Logging Requests to OpenL Tablets Web Services and Their Responses](#)
- [Configuring REST Services Settings](#)
- [Configuring RMI Services Settings](#)
- [Configuring Aegis Databinding](#)
- [Configuring the Instantiation Strategy](#)
- [Configuring the Deployment Filter](#)

Note: There is a specific rule of parsing parameter names in methods. The algorithm checks the case of the second letter in a word and sets the first letter case the same as for the second letter. For example, parameters for `MyMethod` (`String fParam, String sParam`) in REST requests are defined as `FParam` and `sparam`.

Configuring a Data Source

The system supports the following data source implementations:

- [JCR Repository](#)
- [Database Repository](#)
- [File System Data Source](#)
- [Service Exposing Method](#)
- [Amazon AWS S3 Repository](#)
- [Version in deployment name](#)

JCR Repository

This type of the repository is deprecated. Use a database repository instead as described in [Database Repository](#).

To use a JCR repository as a data source, proceed as follows:

5. Locate the `application.properties` file.
By default, this file is stored in the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory.
6. Add the `jcr` value to the `ruleservice.datasource.type` property.
7. Define all JCR repository settings.

The main property in JCR repository settings is `production-repository.factory` that defines the repository access type in one of the following ways:

Repository access type definitions	
Definition	Description
Local repository	<p>Located on the user’s local machine as a folder.</p> <p>The repository factory must be as follows:</p> <pre>production-repository.factory = org.openl.rules.repository.factories.LocalJackrabbitRepositoryFactory</pre> <p>Additional property that defines JCR repository location is as follows:</p> <pre>production-repository.uri = /<OPENL_HOME>/deployment-repository</pre> <p>Note: Only one application can use a local repository at a time. That is, OpenL Tablets Web Services and OpenL Tablets WebStudio cannot be used with a local repository at the same time. If multiple applications need to access a repository, remote access to the repository must be provided for all applications.</p>
Remote repository	<p>Located on a remote server.</p> <p>A recommended way to install and configure a remote repository is as follows:</p> <ol style="list-style-type: none"> Download all sources related to the JCR repository at http://openl-tablets.org/downloads, the Repository ZIP file link. A repository package contains a repository server and an empty JCR repository. Copy the <code>openl-tablets-remote-repository-server-X.X.X.war</code> file from the repository package to the <code><TOMCAT_HOME>\webapps</code> directory. If a secured remote JCR repository is used, define login and password and secret key in the <code>application.properties</code> file. <p>Note: Remember that the Jackrabbit <code>war</code> file must be run before the OpenL Tablets Web Services <code>war</code> file. Tomcat runs <code>war</code> files alphabetically.</p>

A remote repository can be accessed by the following protocols:

Protocols for accessing a remote repository	
Protocol	Description
RMI	<p>To set up access to the repository, edit the <code>bootstrap.properties</code> file located in the <code>jackrabbit</code> folder inside the <code>openl-tablets-remote-server-X.X.X.war</code> file. The <code>bootstrap.properties</code> file contains settings indicating where the repository is located, and the URL which must be used for remote access as follows:</p> <ul style="list-style-type: none"> <code>repository.home={the folder where user’s production repository is located}</code> <code>rmi.url={URL for remote access to the repository}, for example, <code>//localhost:1099/deployment-repository</code></code> <p>The repository factory must be as follows:</p> <pre>production-repository.factory = org.openl.rules.repository.factories.RmiJackrabbitRepositoryFactory</pre> <p>Additional property that defines the remote repository location is as follows:</p> <pre>production-repository.uri = //localhost:1099/deployment-repository</pre>
WebDav	<p>The repository factory must be as follows:</p> <pre>production-repository.factory = org.openl.rules.repository.factories.WebDavRepositoryFactory</pre> <p>Additional property that defines the remote repository location is as follows:</p> <pre>production-repository.uri = http://localhost:8080/deployment-repository</pre>

Security: If a secured remote JCR repository is used, define login and password and secret key in the `application.properties` file.

Attention! A problem can arise if one instance of Tomcat is used for both web archives, that is, `jackrabbit-webapp` and the OpenL Tablets Web Services `war` file. Tomcat will stop working upon startup because of OpenL Tablets Web Services trying to connect to the data source on startup. For the JCR remotely using WebDav case, this means that there are connections by the datasource URL. Tomcat applies such connections and waits until all web applications are deployed. This causes a deadlock, since OpenL Tablets Web Services tries to connect to another application, which cannot respond before OpenL Tablets Web Services is deployed.

To resolve the issue, use one of possible solutions:

1. Use several Tomcat instances, one for `Jackrabbit-webapp`, and another for OpenL Tablets Web Services.
2. Use another application server, such as WebSphere, which supports access to web applications deployed before all other web applications are started.

Database Repository

To use a database repository as a data source, proceed as follows:

1. Add the appropriate driver library for a database.
For example, for MySQL 5.6, it is the `mysql-connector-java-5.1.31.jar` library and it is already located in `webstudio.war`.
2. Locate the `application.properties` file and add the `jcr` value to the `ruleservice.datasource.type` property.
3. In the `application.properties` file set database repository settings as follows.
 - a. Comment the setting `production-repository.factory = org.openl.rules.repository.factories.LocalJackrabbitRepositoryFactory.`
 - b. Define `production-repository.factory = org.openl.rules.repository.db.JdbcDBRepositoryFactory.`
 - c. Set the value for `production-repository.uri` according to the database as follows:

URL value for databases	
Database	URL value
MySQL, MariaD B	<code>jdbc:mysql://[host][:port]/[schema]</code>
Oracle	<code>jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE</code>
MS SQL	<code>jdbc:sqlserver://[serverName[\instanceName][:portNumber]][;property=value[;property=value]]</code>
Postgrge SQL	<code>jdbc:postgresql://[host][:port]/[schema]</code>

For example, for MySQL, `production-repository.uri = jdbc:mysql://localhost:3306/deployment-repository.`

- d. Set login and password for connection to the database defined while installing the database
`production-repository.login` and `production-repository.password.`

Note: The password must be encoded via Base64 encoding schema when `repository.encode.decode.key` is also defined.

```

20  ruleservice.instantiation.strategy.maxthreadsforcompile = 3
21
22  production-repository.factory = org.openl.rules.repository.db.JdbcDBRepositoryFactory
23  production-repository.uri = jdbc:h2:mem:repo;DB_CLOSE_DELAY=-1
24  production-repository.login = root
25  production-repository.password = admin
26  # Secret key for password code/decode
27  repository.encode.decode.key =
28
29  #System Properties

```

Figure 3: Settings for connection to the database production repository in `application.properties`

If a user does not use OpenL Web Studio deploy functionality to locate a project with rules in the database repository, use the `deploy(File zipFile, String config)` method of the `ProductionRepositoryDeployer.class` in the `WEB-INF\lib\org.openl.rules.workspace-5.X.jar` library. The method parameter `zipFile` contains the address to the project zip file and the `config` parameter sets the location of the `deployer.properties` file, containing the same properties as described previously.

File System Data Source

Using a file system as a data source for user projects means that projects are stored in a local folder. This folder represents a single deployment containing all the projects. This is the default data source configured in the system.

To configure a local file system as a data source, proceed as follows:

1. Locate the `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes` directory.
2. In the `application.properties` file, set the `ruleservice.datasource.type` property to `local`. This type of data source does not support deployment and versioning by default.
3. To enable deployment and versioning, do the following:
 - To enable deployment support, set the `ruleservice.datasource.filesystem.supportDeployments` property to `true`.
 - To enable versioning support for deployment, set the `ruleservice.datasource.filesystem.supportVersion` property to `true`.

Users can also pack their rule projects in a `jar` file and use this file as a data source as described in [JAR File Data Source](#).

Note: For proper parsing of Java properties file, the path to the folder must be defined with a slash ("/) as the folders delimiter. Back slash ("\) is not allowed.

Amazon AWS S3 Repository

To use an AWS S3 repository as a data source, proceed as follows:

1. Build a customized version of `WebService` with dependencies on `*org.openl.rules.repository.aws v5.20.54 *`:

Create a `pom.xml` file with the following content:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

```



```

<groupId>com.example.openl</groupId>
<artifactId>webservice-aws</artifactId>
<packaging>war</packaging>
<version>1.0-beta</version>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <org.openl.version>5.21.5</org.openl.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.openl.rules</groupId>
    <artifactId>org.openl.rules.repository.aws</artifactId>
    <version>${org.openl.version}</version>
  </dependency>
  <dependency>
    <groupId>org.openl.rules</groupId>
    <artifactId>org.openl.rules.ruleservice.ws</artifactId>
    <type>war</type>
    <version>${org.openl.version}</version>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.9.5</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-annotations</artifactId>
      <version>2.9.5</version>
    </dependency>
    <dependency>
      <groupId>commons-codec</groupId>
      <artifactId>commons-codec</artifactId>
      <version>1.11</version>
    </dependency>
  </dependencies>
</dependencyManagement>
</project>

```

2. Locate the <TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\classes directory, and add the following properties in the application.properties file:

```

ruleservice.datasource.type = jcr
production-repository.factory = org.openl.rules.repository.aws.S3Repository
production-repository.bucket-name = yourBucketName
production-repository.region-name = yourS3Region
production-repository.access-key = yourAccessKey
production-repository.secret-key = yourSecretKey

```

Version in deployment name

If the Deployment repository was created in an OpenL version older than 5.20, the option “Version in deployment name” must be enabled for backward compatibility.

The 5.20 version of the OpenL Deployment Repository contains only actual deployments which are exposed as services. Each new deployment updates the current deployment, while older versions are hidden in history and cannot be loaded into the RuleService directly. Different API versions of services are located in the different deployments. They are distinguished by a suffix generated in WebStudio according to the API version in rules-

deploy.xml. As a result, services are exposed more quickly. However, if you created a repository in a version of OpenL older than 5.20, and migrate to a newer webservices, you must enable option “Version in deployment name,” otherwise your services will not be exposed correctly.

In this case, add the following property in the application.properties file:

```
version-in-deployment-name = true
```

If you create a new repository, omit this property or set it to false.

Service Exposing Method

Service exposing method specifies a method to expose user’s OpenL Tablets Web Services.

Common flow of service exposing is as follows:

1. Retrieve service descriptions that must be deployed using Service Configurer.
2. Undeploy the currently running services that are not in services defined by Service Configurer.
Some services can become unnecessary in the new version of the product.
3. Redeploy currently running services that are still in services defined by Service Configurer, such as service update.
4. Deploy new services not represented earlier.

To set the method of exposing services, configure a Spring bean with the `ruleServicePublisher` name in `openl-ruleservice-publisher-beans.xml`.

This bean supports mapping a concrete publisher for service configuration or uses a default publisher if the publisher is not defined in the OpenL Tablets project.

To implement a new publisher, use any framework by implementations of `org.openl.rules.ruleservice.publish.RuleServicePublisher` interface and register it in the `ruleServicePublisher` bean or use one of the following predefined implementations:

- [CXF Web Services Implementation](#)
- [CXF REST Service implementation](#)
- [RMI Service Implementation](#)

CXF Web Services Implementation

CXF Web Service Publisher implementation class is

`org.openl.rules.ruleservice.publish.JAXWSRuleServicePublisher`. The Spring configuration for CXF Web Service Publisher is located in `openl-ruleservice-webservice-publisher-beans.xml` file.

Note: The full web service address is `webserver_context_path/ws_app_war_name/address_specified_by_you`.

CXF REST Services Implementation

CXF REST Service Publisher implementation class is

`org.openl.rules.ruleservice.publish.JAXRSRuleServicePublisher`. The Spring configuration for this publisher is located in the `openl-ruleservice-jaxrs-publisher-beans.xml` file.

Note: The full web service address is

`webserver_context_path/ws_app_war_name/REST/address_specified_by_you` or `webserver_context_path/ws_app_war_name/address_specified_by_you`. It depends on project configuration.

RMI Services Implementation

RMI Service Publisher implementation class is

`org.openl.rules.ruleservice.publish.RmiRuleServicePublisher`. The Spring configuration for this publisher is located in the `openl-ruleservice-rmi-publisher-beans.xml` file.

Note: The full RMI service address is `rmi://hostname:port/address_specified_by_you`.

Configuring System Settings

There are several options extending rules behavior in OpenL Tablets:

- Custom Spreadsheet Type
- Rules Dispatching Mode
- Table Dispatching Validation Mode

These settings are defined in the `application.properties` configuration file.

Configuring a Number of Threads to Rules Compilation

The system supports parallel rules compilation. Rules compilation consumes a large amount of memory. If the system tries to compile too many rules at once, it fails with an out of memory exception.

The setting that limits the amount of threads to compile rules is defined in the `application.properties` file.

By default, only three threads can compile rules in parallel:

```
ruleservice.instantiation.strategy.maxthreadsforcompile = 3
```

For example, to permit only one thread to compile rules, set value to one as follows:

```
ruleservice.instantiation.strategy.maxthreadsforcompile = 1
```

Logging Requests to OpenL Tablets Web Services and Their Responses

The system provides an ability to log all requests to OpenL Tablets Web Services and their responses.

The setting is defined in the `application.properties` file.

By default, logging is disabled:

```
ruleservice.logging.enabled = false
```

To enable logging, set `ruleservice.logging.enabled = true`.

OpenL Tablets Web Services supports storing requests and responses for SOAP and REST publishers in an external storage. This feature supports using any external data source for storing log data or using the Apache Cassandra database out of the box.

To enable this feature, set `ruleservice.logging.store.enabled = true`.

For each request to OpenL Tablets Web Services, the system creates a logging record, which is populated with data during request processing and then can be stored in the required external storage. For logging records, the `org.openl.ruleservice.logging.LoggingInfo` class is used. It contains the following data:

org.openl.ruleservice.logging.LoggingInfo class data	
Field name	Description
requestMessage	Contains request data for logging, such as request body, URL, request header, and request content type.
responseMessage	Contains response data for logging, such as response body, response status, and response header.
incomingMessageTime	Time when request is received by the server.
outcomingMessageTime	Time when response message preparing is complete and the message is ready to be sent to the client.
service	OpenL Tablets service used for the call. Data includes service name, compiled OpenL Tablets rules, and other information.
inputName	Method used for the call.
parameters	Parameters of the call, which is an array of objects after binding request message to models.

When the logging data is collected, the system invokes the storing service responsible for saving logging data. The storing service must implement the `org.openl.ruleservice.logging.LoggingInfoStoringService` interface.

The Apache Cassandra database can be used as external storage. The first step for using Cassandra is to download the zip archive for the applicable version of OpenL Tablets from this repository, <https://repo1.maven.org/maven2/org/openl/rules/org.openl.rules.ruleservice.ws.logging.cassandra/>, and unzip it to the folder `webapps/<web services war file name>/WEB-INF/lib`.

Cassandra connection settings are defined in the `application.properties` file.

Connection settings defined in the <code>cassandra.properties</code> file	
Property name	Description
<code>cassandra.contactpoint</code>	Connection points to the Cassandra node.
<code>cassandra.port</code>	Port.
<code>cassandra.keyspace</code>	Keyspace to be used.

Before creating a schema in Cassandra, keyspace must be created as described in https://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_keyspace_r.html.

To create a schema in the Cassandra database, start OpenL Tablets Web Services with the `ruleservice.logging.store.type.cassandra.schema.create = true` property. When the schema is created, set this property to the `false` value; otherwise, the application tries to recreate the schema and fails, because the schema is already created.

As a result, the following table with the `LoggingRecord` name is created in the Cassandra database:

LoggingRecord table created in the Cassandra database		
Column name	Type	Description
ID	TEXT	Unique ID for the request. It is a primary key for the record.
INCOMINGTIME	TIMESTAMP	Incoming request time.
OUTCOMINGTIME	TIMESTAMP	Outgoing response time.
REQUEST	TEXT	Request body.
RESPONSE	TEXT	Response body.

LoggingRecord table created in the Cassandra database		
Column name	Type	Description
INPUTNAME	TEXT	Input name.
URL	TEXT	URL of the request.

Configuring REST Services Settings

The system supports using HTTP 200 Status for all RESTful services requests.

The setting is defined in the `application.properties` file.

By default, this feature is disabled:

```
ruleservice.jaxrs.responseStatusAlwaysOK = false
```

To enable this feature, set `ruleservice.jaxrs.responseStatusAlwaysOK = true`.

Configuring RMI Services Settings

The appropriate port and host name for RMI can be defined in the `application.properties` file.

By default, these settings are defined as follows:

```
ruleservice.rmiPort = 1099 // Port for RMI
ruleservice.rmiHost = 127.0.0.1 // Used as host for RMI
```

Configuring Aegis Databinding

The system provides an ability to configure Aegis databinding settings.

For more information on Aegis databinding, see CXF Aegis databinding documentation.

The setting is defined in the `application.properties` file.

The default settings are as follows:

```
ruleservice.aegisbinding.readXsiTypes = true
ruleservice.aegisbinding.writeXsiTypes = true
ruleservice.aegisbinding.ignoreNamespaces = false
```

Configuring the Instantiation Strategy

The system provides an ability to select an instantiation strategy.

The setting is defined in the `application.properties` file.

By default, the lazy initialization strategy is enabled:

```
ruleservice.instantiation.strategy.lazy = true
```

Modules are compiled upon the first request and can be unloaded in future for memory save.

To disable the lazy initialization strategy, set `ruleservice.instantiation.strategy.lazy = false`. All modules are compiled on the application launch.

Configuring the Deployment Filter

The system provides the ability to set up the Deployment Filter to select deployments for installation in Web Services from the datasource repository when several applications use the same datasource repository. Filtering selects deployments by deployment name.

The settings are defined in the `application.properties` file.

By default, the Deployment Filter is disabled:

```
ruleservice.datasource.deployments =
```

To enable the Deployment Filter, set the exact deployment names using a comma separator, or use the wildcard character to enable the filter to match patterns in the deployment name:

```
ruleservice.datasource.deployments = foo-deployment, bar-*
```

The wildcard character “*” matches any characters in the deployment name. You can use multiple asterisks.

- Single: any of these single wildcard character patterns will detect `foo-deployment: foo-*, *deployment`
- Multiple: any of these single wildcard character patterns will detect `foo-deployment: *deploy*, *deployment*`

5 OpenL Tablets Web Services Customization

This section introduces general OpenL Tablets Web Services customization algorithm and explains the following available customization points:

- [OpenL Tablets Web Services Customization Algorithm](#)
- [Service Configurer](#)
- [Multimodule with Customized Dispatching](#)
- [Dynamic Interface Support](#)
- [Interface Customization through Annotations](#)
- [JAR File Data Source](#)
- [Data Source Listeners](#)
- [Service Publishing Listeners](#)
- [Variations](#)

5.1 OpenL Tablets Web Services Customization Algorithm

If a project has specific requirements, OpenL Tablets Web Services customization algorithm is as follows:

1. Create a new Maven project that extends OpenL Tablets Web Services.
2. Add or change the required points of configuration.
3. Add the following dependency to the `pom.xml` file with the version used in the project specified:

```
<dependency>
  <groupId>org.openl.rules</groupId>
  <artifactId>org.openl.rules.ruleservice.ws</artifactId>
  <version>5.X.X</version>
  <type>war</type>
  <scope>runtime</scope>
</dependency>
```

4. Use the following Maven plugin to control the Web Application building with user's custom configurations and classes:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <warSourceDirectory>webapps/ws</warSourceDirectory>
    <!--Define war name here-->
    <warName>${war.name}-${project.version}</warName>
    <packagingExcludes>
      <!--Exclude unnecessary libraries from parent project here-->
      WEB-INF/lib/org.openl.rules.ruleservice.ws.lib-*.jar
    </packagingExcludes>
    <!--Define paths for resources. Developer has to create a file with the same
    name to overload existing file in the parent project-->
    <webResources>
      <resource>
        <directory>src/main/resources</directory>
      </resource>
      <resource>
        <directory>war-specific-conf</directory>
      </resource>
    </webResources>
  </configuration>
```

```
</plugin>
```

Customization points are described in the following table:

Customization points	
Point	Description
Service Configurer	Defines all services to be exposed and modules contained in each service.
Multimodule with Customized Dispatching	Provides a possibility to handle dispatching between modules.
Dynamic Interface Support	Generates an interface for services at runtime.
Interface Customization through Annotations	
JAR File Data Source	Pack user's rule projects into a <code>jar</code> file and places the archive in the classpath.
Data Source Listeners	
Variations	Additional calculation of the same rule with a slight modification in its arguments.

5.2 Service Configurer

This section introduces Service Configurer and includes the following topics:

- [Understanding Service Configurer](#)
- [Deployment Configuration File Used by Service Configurer](#)
- [Service Description](#)

Understanding Service Configurer

Service Configurer defines all services to be exposed, such as modules contained in each service, the service interface, and runtime context provision.

Modules for a service can be retrieved for different projects. Each deployment contained in a data source has a set of properties and can be represented in several versions. Deployment consists of projects that also have properties and contains some modules. There can be only one version of a specific project in the deployment.

Each module for a service can be identified by the deployment name, deployment version, project name inside the deployment, and module name inside the module.

Users can implement different module gathering strategies according to their needs. Users can choose deployments and projects with concrete values of a specific property, such as service for some LOB or service containing modules with an expiration date before a specific date, or versions of deployments, or both these approaches.

OpenL Tablets users typically need web services containing several rule projects or modules. In this case, multiple modules can be united in one service using a simple service description. Service description contains all information about the required service, such as the service name, URL, all modules that form the service, and the service class, and can be expanded to contain new configurations. To instantiate several modules, users can rely on the OpenL MultiModule mechanism that combines a group of modules as a single rules engine instance.

Deployment Configuration File Used by Service Configurer

By default, OpenL Tablets Web Services uses `LastVersionProjectsServiceConfigurer` which deploys last version projects from deployments. This implementation uses the service configuration `rules-deploy.xml` file from the project root folder. This file can be created manually or via OpenL Tablets WebStudio. An example of the `rules-deploy.xml` file is as follows:

```
<rules-deploy>
  <isProvideRuntimeContext>true</isProvideRuntimeContext>
  <isProvideVariations>false</isProvideVariations>
  <serviceName>myService</serviceName>
  <serviceClass>com.example.MyService </serviceClass>
  <url>com.example.MyService</url>
  <publishers>
    <publisher>RESTFUL</publisher>
    <publisher>WEBSERVICE</publisher>
  </publishers>
  <configuration>
    <entry>
      <string>someString</string>
      <string>someString</string>
    </entry>
  </configuration>
</rules-deploy>
```

Project configuration		
Tag	Description	Required
<code>isProvideRuntimeContext</code>	Identifies, if set to <code>true</code> , that a project provides a runtime context. The default value is defined in the <code>application.properties</code> file.	No
<code>isProvideVariations</code>	Identifies, if set to <code>true</code> , that a project provides variations. The default value is defined in the <code>application.properties</code> file.	No
<code>serviceName</code>	Defines a service name.	No
<code>serviceClass</code>	Defines a service class. If it is not defined, a generated class is used.	No
<code>version</code>	Defines a service version.	No
<code>url</code>	Defines URL for a service.	No
<code>annotationTemplateClassName</code>	Defines an interface being used as a template to annotate dynamic generated class.	No
<code>groups</code>	Defines a list of comma-separated groups used for this project as described in Understating Groups Annotations .	No
<code>publishers</code>	Defines a list of publishers for a project.	No
<code>configuration</code>	Is used as extension point for custom service configuration.	No
<code>lazy-modules-for-compilation</code>	Defines a list of modules to be loaded in case lazy loading mechanism is used. Module names can contain Ant path expressions.	No

Service Description

Commonly each service is represented by rules and the service interface and consists of the following elements:

Service description	
Service	Description
Service name	Unique service identifier.
Service URL	URL path for the service. It is absolute for the console start and relative to the context root for the <code>ws.war</code> case.
Service class	Interface of the service to be used at the server and the client side.
Version	Number of the service version.
Rules	Module or a set of modules to be combined together as a single rules module.
Provide runtime context flag	Identifier of whether the runtime context must be added to all rule methods. If it is set to <code>true</code> , the <code>IRulesRuntimeContext</code> argument must be added to each method in the service class.
Support variations flag (optional)	Identifier of whether the current service supports variations. For more information on variations, see Variations .

Users can create their own implementation of Service Configurer interface

`org.openl.rules.ruleservice.conf.ServiceConfigurer` and register it as a Spring bean with the `serviceConfigurer` name, or use one of the following implementations provided by OpenL Tablets Web Services:

- `org.openl.rules.ruleservice.conf.SimpleServiceConfigurer`
It is designed for use with a data source having one deployment. It exposes deployment and creates service for one predefined project in this deployment.
- `org.openl.rules.ruleservice.conf.LastVersionProjectsServiceConfigurer`
It exposes deployments based on the last version and creates one service for each project in the deployment. It reads configuration for service deployment from `rules-deploy.xml` in a project.

5.3 Multimodule with Customized Dispatching

There is an additional mode for the multimodule which allows handling dispatching between modules by user's own logic. That means OpenL Tablets passes the control of selection of the needed module to user's own class.

To adjust multimodule with user's own dispatching, proceed as follows:

1. Create Java interface representing the required rules.
2. For each method from the interface, determine the dispatching:
 - For methods that represents **Data tables**, provide implementation of `org.openl.rules.ruleservice.publish.cache.dispatcher.ModuleDispatcherForData` and mark that method by the `org.openl.rules.ruleservice.publish.cache.dispatcher.DispatchedData` annotation.
 - For methods that represent **Rules**, provide implementation of `org.openl.rules.ruleservice.publish.cache.dispatcher.ModuleDispatcherForMethods` and mark that method by the `org.openl.rules.ruleservice.publish.cache.dispatcher.DispatchedMethod` annotation.
3. Create implementation of `org.openl.rules.ruleservice.publish.RuleServiceInstantiationStrategyFactory` that returns

`DispatchedMultiModuleInstantiationStrategy` instead of a lazy multimodule, by default, and register it in `openl-ruleservice-override-beans.xml`.

Notes:

- `ModuleDispatcherForData` and `ModuleDispatcherForMethods` must have a public constructor without parameters. The aim of these classes is to select the needed `Module` according to the Runtime context and the executed method. This means the rule name and arguments for the method representing **Rule**, and **Data table** for the method representing data.
- If a dispatched multimodule is used, the interface with the annotated methods is obligatory, otherwise an exception is given.
- If a getter and setter for specific Data is available simultaneously, only one of them can be annotated.
- Different dispatching logic for different methods can be provided.
- See example in `org.openl.rules.ruleservice.multimodule.DispatchedMultiModuleTest`.

5.4 Dynamic Interface Support

OpenL Tablets Web Services supports interface generation for services at runtime. This feature is called Dynamic Interface Support. If static interface is not defined for a service, the system generates it. The system uses an algorithm that generates an interface with all methods defined in the module or, in case of a multimodule, in the list of modules.

This feature is enabled by default. To use a dynamic interface, do not define a static interface for a service.

It is not a good practice to use all methods from a module in a generated interface because of the following limitations:

- All return types and method arguments in all methods must be transferrable through network.
- An interface for web services must not contain the method designed for internal usage.

The system provides a mechanism for filtering methods in modules by including or excluding them from the dynamic interface.

This configuration can be applied to projects using the `rules.xml` file. An example is as follows:

```
<project>
  <!-- Project name. -->
  <name>project-name</name>
  <!-- OpenL project includes one or more modules. -->
  <modules>
    <module>
      <name>module-name</name>

      <!--
      Rules root document. Usually excel file on file system.
      -->
      <rules-root path="rules/Calculation.xlsx"/>
      <method-filter>
        <includes>
          <value>.*determinePolicyPremium.*</value>
          <value>.*vehiclePremiumCalculation.*</value>
        </includes>
      </method-filter>
    </module>
  </modules>
```

```

    <!-- Project's classpath. -->
    <classpath>
      <entry path="lib/*"/>
    </classpath>
  </project>

```

For filtering methods, define the `method-filter` tag in the `rules.xml` file. This tag contains `includes` and `excludes` tags. The algorithm is as follows:

- If the `method-filter` tag is not defined in the `rules.xml`, the system generates a dynamic interface with all methods provided in the module or modules for multimodule.
- If the `includes` tag is defined for method filtering, the system uses the methods which names match a regular expression of defined patterns.
- If the `includes` tag is not defined, the system includes all methods.
- If the `excludes` tag is defined for method filtering, the system uses methods which method names do not match a regular expression for defined patterns.
- If the `excludes` tag is not defined, the system does not exclude the methods.

If OpenL Tablets Dynamic Interface feature is used, a client interface must also be generated dynamically at runtime. Apache CXF supports the dynamic client feature. For more information on dynamic interface support by Apache CXF, see <http://cxf.apache.org/docs/dynamic-clients.html>.

Note: If a project is empty (does not contain any method), it will not be available as a service.

5.5 Interface Customization through Annotations

This section describes interface customization using annotations. The following topics are included:

- [Interceptors for Service Methods](#)
- [Annotation Customization for Dynamic Interfaces](#)
- [Understandings Groups in Annotations](#)

Interceptors for Service Methods

Interceptors for service methods can be specified using the following annotations:

- `@ServiceCallBeforeInterceptor`
This method annotation is defined before interceptors and the array of interceptors must be registered in the annotation parameter. All interceptors must implement the `org.openl.rules.ruleservice.core.interceptors.ServiceMethodBeforeAdvice` interface. The main goal of these interceptors is adding additional logic before service method invocation, such as validation for service method arguments or using additional logic of input types convention.
- `@ServiceCallAroundInterceptor`
This method annotation defines around interceptors and the array of interceptors must be registered in the annotation parameter. All interceptors must implement the `org.openl.rules.ruleservice.core.interceptors.ServiceMethodAroundAdvice` interface. This type of interceptors is used for adding around logic for service method invocation. An example is when the arguments of the case service method must be converted to another type before using them in service rules, and the results also require additional processing before return.
- `@ServiceCallAfterInterceptor`

This method annotation is defined after interceptors and the array of interceptors must be registered in the annotation parameter. This type of interceptions can be used for result processing or error handling before return by service method.

There are two types of after interceptors:

Inceptor types for the <code>@ServiceCallAfterInterceptor</code> annotation	
Inceptor	Description
After Returning	Intercepts the result of a successfully calculated method, with a possibility of post processing of the return result, including result conversion to another type. In this case, the type must be specified as the return type for the method in the service class. The <code>After Returning</code> interceptors must inherit <code>org.openl.rules.ruleservice.core.interceptors.AbstractServiceMethodAfterReturningAdvice</code> .
After Throwing	Intercepts a method that has an exception thrown, with a possibility of post processing of an error and throwing another type of exception. The <code>After Returning</code> interceptors must inherit <code>org.openl.rules.ruleservice.core.interceptors.AbstractServiceMethodAfterThrowingAdvice</code> .

- `@ServiceExtraMethod`

This method annotation defines a proxy for the extra method absent in OpenL rules. The proxy must implement `org.openl.rules.ruleservice.core.annotations.ServiceExtraMethodHandler` interface, and it exposes methods that differ in signature with the rules or do not exist in the Excel sheet.

For example:

Excel contains `String hello(String)` method and this method should be proxied via `String hello(Integer)`.

The proxy class has access to the classes generated for OpenL Datatypes. It can be used when creating mapping between OpenL model and external model, for example:

```
public static class LoadClassExtraMethod implements ServiceExtraMethodHandler<Object> {
    @Override
    public Object invoke(Method interfaceMethod, Object serviceBean, Object... args)
throws Exception {
        // MyBean is Datatype defined in OpenL
        Class<?> myBeanClass = Thread.currentThread().getContextClassLoader()
            .loadClass("org.openl.generated.beans.MyBean");
        Object myBean = myBeanClass.newInstance();
        // ... Do some mapping below and then return result
        return myBean;
    }
}
```

Note: Java byte code does not store arguments names in interfaces, so they have names like 'arg0', 'arg1', etc. If you want request parameters to have more meaningful names, use `@Name` annotation. This annotation is used together with `@ServiceExtraMethod` to define “pretty” names REST services instead of 'arg0', 'arg1'...

Annotation Customization for Dynamic Interfaces

Annotation customization can be used for dynamically generated interfaces. This feature is only supported for projects that contain the `rules-deploy.xml` file. To enable annotation customization, proceed as follows:

1. Add the `annotationTemplateName` tag to `rules-deploy`.

An example is as follows:

```
<rules-deploy>
  <isProvideRuntimeContext>true</isProvideRuntimeContext>
  <isProvideVariations>false</isProvideVariations>
  <serviceName>dynamic-interface-test3</serviceName>
  <annotationTemplateName>org.openl.ruleservice.dynamicinterface.test.MyTemplateClass</annotationTemplateName>
  <url></url>
</rules-deploy>
```

2. Define a template interface with the annotated methods with the same signature as in the generated dynamic interface.

This approach supports replacing argument types in the method signature with types assignable from the generated interface. For example, consider the following methods in the generated dynamic interface:

```
void someMethod(IRulesRuntimeContext context, MyType myType);
void someMethod(IRulesRuntimeContext context, OtherType otherType);
```

Add an annotation to the first method using a signature in the template interface as follows:

```
@ServiceCallAfterInterceptor(value = { MyAfterAdvice.class })
void someMethod(IRulesRuntimeContext context, MyType myType);
```

If `MyType` is generated in the runtime class, use a type that is assignable from the `MyType` class.

An example is as follows:

```
@ServiceCallAfterInterceptor(value = { MyAfterAdvice.class })
void someMethod(IRulesRuntimeContext context, @AnyType(".*MyType") Object myType);
```

Note that this example uses the `@AnyType` annotation. If this annotation is skipped, this template method is applied to both methods, because `Object` is assignable from both types `MyType` and `OtherType`.

The `@AnyType` annotation value is a Java regular expression of a canonical class name. Use this annotation if more details are required to define a template method.

Note: A user also can use class level annotations for a dynamically generated class. It can be useful for JAXWS or JAXRS interface customization.

Understandings Groups in Annotations

The **Groups in annotations** feature allows developers to use annotations for intercepting service methods. For more information on intercepting service methods, see [Interceptors for Service Methods](#).

The system uses such markup for interceptor in cases that require additional control for interface intercepting. Usually it makes sense in requirements when one OpenL Rules project is used by two or more Rule Service-based applications at the same time. For example, there are two applications using one repository with the same projects, where the first application requires logging invocations to a database, and this logic is implemented via interceptors, and the second application does not need this logic and misses classes implementing this logic in the classpath. The new feature allows both applications to use the same project without failure. In this example, all interceptors implementing logging logic must be marked up with a separate group, and all marked interceptors are skipped by the appropriate application.

To support more than one interceptor type for a method, the following annotations are introduced:

- `ServiceCallBeforeInterceptors`
- `ServiceCallAroundInterceptors`
- `ServiceCallAfterInterceptors`

These annotations must be declared with an appropriate array of annotations as described in [Interceptors for Service Methods](#).

By default, the system uses interceptors that belong to any group. To configure the system to use specific groups, modify `*` in the `ruleservice.datasource.groups` property. These groups will be used for projects that have no project level definition for groups. To define groups on the project level, use the `rules-deploy.xml` project deployment file.

Note: Use comma separated groups if more than one group must be used at the same time. The following groups are supported:

Group types for interceptor annotations	
Group	Description
All	This group is used to mark the interceptor to be used by all systems. This is a default option and all interceptors belong to this group if group markup is missing.
RULESERVICE, WEBSERVICE, GROUP1, GROUP2, GROUP3	Logically these groups are the same, but it is recommended to use the WEBSERVICE group for the OpenL Tablets Web Services application and the RULESERVICE group for an embedded solution.

5.6 JAR File Data Source

If rule projects and the `rules.xml` project descriptor are packed into a JAR file and placed in the classpath, these projects are deployed in the configured data source at the application launch.

Proceed as follows:

1. Put the JAR file with the project to `<TOMCAT_HOME>\webapps\<web services file name>\WEB-INF\lib`.
2. In the `application.properties` file, set up the `ruleservice.datasource.deploy.classpath.jars = true`

Note: Project deployment is skipped if the data source already contains the project with the same name.

5.7 Data Source Listeners

A data source registers data source listeners and notifies some components of the OpenL Tablets Web Services frontend about modifications. The only available event type on the production repository modification is about newly added deployment.

A service manager is always a data source listener because it must handle all modifications in the data source.

Users can add their own listener implementing `org.openl.rules.ruleservice.loader.DataSourceListener` for additional control of data source modifications with the required behavior and register it in data source.

5.8 Service Publishing Listeners

The OpenL Tablets Rule Service solution uses

`org.openl.rules.ruleservice.publish.MultipleRuleServicePublisher` class implementation as a main publisher to support multiple publishers in the system. This publisher is configured as the following Spring bean:

```

<bean id="ruleServicePublisher"
class="org.openl.rules.ruleservice.publish.MultipleRuleServicePublisher">
  <property name="defaultRuleServicePublishers">
    <list>
      a list of all supported publishers by default (For example SOAP, REST,
RMI if publisher type is not defined for project)
    </list>
  </property>
  <property name="supportedPublishers">
    <map>
      <entry key="WEBSERVICE" value-
ref="webServiceRuleServicePublisher"/>
      <entry key="RESTFUL" value-
ref="JAXRSServicesRuleServicePublisher"/>
      <entry key="RMI" value-ref="RMIServicesRuleServicePublisher"/>
    </map>
  </property>
  <property name="listeners">
    <bean class="org.openl.rules.ruleservice.conf.BeanCollectionFactoryBean">
      <property name="beanType"
value="org.openl.rules.ruleservice.publish.RuleServicePublisherListener" />
    </bean>
  </property>
</bean>

```

Service publishing listeners notify about the deployed or un-deployed OpenL Tablets projects. Users can add their own listeners implementing

`org.openl.rules.ruleservice.publisher.RuleServicePublisherListener` for additional control of deploying and un-deploying projects with the required behavior and add them to Spring configuration. The `org.openl.rules.ruleservice.conf.BeanCollectionFactoryBean` bean automatically finds all listeners in the Spring context and registers them in the service publisher.

The `org.openl.rules.ruleservice.publisher.RuleServicePublisherListener` interface has the following methods:

Method in <code>org.openl.rules.ruleservice.publisher.RuleServicePublisherListener</code>	
Inceptor	Description
<code>onDeploy(OpenLService)</code>	Invoked each time when the OpenL Tablets service is deployed with the publisher that fires this listener.
<code>onUndeploy(String serviceName)</code>	Invoked each time when the service with the defined name is un-deployed.

5.9 Variations

In highly loaded applications, performance of execution is a crucial point in development. There are many approaches to speed up the application. One of them is to calculate rules with variations.

A **variation** stands for additional calculation of the same rule with a slight modification in its arguments. Variations are very useful when a rule must be calculated several times with similar arguments. The idea of this approach is to once calculate rules for particular arguments and then recalculate only the rules or steps that depend on the modified, by variation, fields in those arguments.

The following topics are included:

- [Variation Algorithm](#)
- [Predefined Variations](#)

- [Variations Factory](#)
- [Variations as Rules](#)
- [Example](#)

Variation Algorithm

A rule that can be calculated with variations must have the following methods in a service class:

- original method with a corresponding rule signature
- method with injected variations

The method enhanced with variations has a signature similar to the original method. Add the argument of the `org.openl.rules.variation.VariationsPack` type as the last argument. The return type must be generic `org.openl.rules.variation.VariationsResult<T>`, where `T` is the return type of the original method.

The `VariationsPack` class contains all required variations to be calculated. The `VariationsResult<T>` class contains results of the original calculation, without any modifications of arguments, and all calculated variations that can be retrieved by variation ID. There can be errors during calculation of a specific variation. There are two methods to get result of a particular variation:

Methods for getting result of a particular variation	
Method	Description
<code>getResultForVariation(String variationID)</code>	Returns the result of a successfully calculated variation.
<code>getFailureErrorForVariation(String variationID)</code>	Returns the corresponding error message

Note: When using a user's own service class instead of the one generated by default, the original method must be defined for each method with variations.

Note: The result of original calculation can be retrieved in the same manner as for all variations, by using the special 'Original calculation' ID in code as `org.openl.rules.project.instantiation.variation.NoVariation.ORIGINAL_CALCULATION`.

Predefined Variations

A variation typically has a unique ID and is responsible for modifying arguments and restoring original values. The ID is a `String` value used to retrieve the result of the calculation with this variation.

By default, the variation's abstract class `org.openl.rules.project.instantiation.variation.Variation` has two methods, `applyModification` and `revertModifications`. The first method modifies arguments; the second rolls back the changes. For this purpose, a special instance of `Stack` is passed to both these methods: in the `applyModification` method, the previous values must be stored; in `revertModifications`, the previous values can be retrieved from the `Stack` and saved into arguments.

There are several types of predefined variations in the `org.openl.rules.variation` package:

Predefined variation types in the <code>org.openl.rules.variation</code> package	
Variation type	Description
<code>NoVariation</code>	Empty variation without any modifications. It is used for the original calculation and has a predefined 'Original calculation' ID.

Predefined variation types in the <code>org.openl.rules..variation</code> package	
Variation type	Description
<code>ArgumentReplacementVariation</code>	Variation that replaces an entire argument. It was introduced because <code>JXPathVariation</code> cannot replace a value of a root object, or argument. The argument index, value to be set instead of the argument, and ID are required to construct this variation.
<code>JXPathVariation</code>	Variation that modifies an object field or replaces an element in the array defined by the special path. XPath is used to analyze paths and set values to corresponding fields, therefore use XPath-consistent path expressions. The following data is required for this variation: <ul style="list-style-type: none"> • index of the argument to be modified • path to the field that must be modified in the XPath notation • value to be set instead of the original field value • ID For more information on XPath, see http://commons.apache.org/jxpath/ .
<code>ComplexVariation</code>	Variation that combines multiple variations as a single variation. It is applicable when different fields or arguments must be modified.
<code>DeepCloningVariation</code>	Variation used to avoid reverting changes of a specific variation that will be delegated to <code>DeepCloningVariation</code> . This variation clones user's arguments and thus allows avoiding any problems caused by changes in arguments. This variation is not recommended because of performance drawbacks: the argument cloning takes time so the variations usage can be useless.

If predefined implementations do not satisfy user needs, implement user's own type of variation that inherits the `org.openl.rules..variation.Variation` class. Custom implementations can be faster than the predefined variations in case they use direct access to fields instead of a reflection as in `JXPathVariation`.

Note: Data binding for custom implementations of variation must be provided to pass the variations through SOAP in OpenL Tablets Web Services.

Variations Factory

The `org.openl.rules.project.VariationsFactory` class is a utility class for simple creation of predefined variations. It uses the following arguments:

Variations factory arguments	
Argument	Description
<code>variationId</code>	Unique ID for a variation.
<code>argumentIndex</code>	0-based index of an argument to be modified.
<code>path</code>	Path to the field to be modified, or just a dot <code>.</code> to modify the root object, that is, the argument.
<code>valueToSet</code>	Value to set by path.
<code>cloneArguments</code>	Identifier of whether cloning must be used.

Usually `VariationsFactory` creates the `JXPathVariation` variation which covers most cases of variations usage. When a dot `.` is specified as a path, `ArgumentReplacementVariation` is constructed. The `cloneArguments` option says to `VariationsFactory` to wrap created variation by `DeepCloningVariation`.

An alternative way is to use a special `VariationDescription` bean that contains all fields described previously in this section. It is useful to transmit a variation in OpenL Tablets Web Services and define variations in rules.

Variations as Rules

The process of determining the variations is a kind of decision making process and can be represented in the form of rules. A user can write rules that define variations according to the input arguments. Then such rules are used as a variations provider during execution. That means, the OpenL engine passes an argument to the particular rule that has the same signature as the rule to be calculated with variations that returns a set of variation descriptions that will be used to create variations.

To write variations in rules, proceed as follows:

1. Define a special rule that returns `VariationDescription[]` and takes arguments similar to the method that must be calculated with variations.

All `VariationDescriptions` returned from the rule are passed to `VariationsFactory` as described in `Variations Factory` to construct variations and add them to initial `VariationsPack`.

2. Add the import of `org.openl.rules.variation` package into the Environment Table to make `VariationDescription` available for rules.

3. Mark methods with variations by the special annotation

```
@VariationsFromRules(ruleName = "<name of the rule from the first step>")
```

Warning: The method for retrieving variations must be defined in a service class.

4. Enable variations in Service Configurer and data binding.

By default, there is the `ruleservice.isSupportVariations` option in `openl-ruleservice.properties` that must be set to `true`. It is passed to Service Configurer to create services with variations support and to `AegisDatabindingConfigurableFactoryBean` that registers bindings for all predefined variation classes.

Note: When methods with `@VariationsFromRules` annotation are called, `VariationsPack` can be null. In this case, only variations from rules are used. Otherwise, if a non-null `VariationsPack` is provided in the arguments, all variations are calculated: the variations from rules and from `VariationsPack` in arguments.

Example

Consider rules that calculate premium for a policy:

```
Spreadsheet SpreadsheetResult processPolicy(Policy policy)
```

There is a special method for variations from rules:

```
Method VariationDescription[] processPolicyVariations(Policy policy)
```

To calculate premium with variations from rules, the service class must contain the following methods:

- `SpreadsheetResult processPolicy(Policy policy); //original method.`
- `@VariationsFromRules(ruleName = "processPolicyVariations")`
- `VariationsResult<SpreadsheetResult> processPolicy(Policy policy, VariationsPack variations); //method enhanced with variations.`
- `VariationDescription[] processPolicyVariations(Policy policy); //method for retrieving the variations from rules.`

6 Appendix A: Tips and Tricks

This appendix provides useful additional information on OpenL Tablets Web Services usage and customization and includes the following topics:

- [Using OpenL Tablets Web Services from Java Code](#)
- [Using OpenL Tablets REST Services from Java Code](#)

6.1 Using OpenL Tablets Web Services from Java Code

This section illustrates how to write a client code that invokes OpenL Tablets Web Services projects. Another way can be used to invoke services but it is recommended to use Apache CXF framework to prevent additional effort for data binding.

A project in OpenL Tablets Web Services can be exposed via a static interface or dynamic interface generated in runtime. A client code is different in each case. If the project uses a static interface, use the `ClientFactoryBean` class from CXF. For more information on using CXF for a static interface, see CXF documentation.

The following example illustrates client code generation for the `MyClass` static interface:

```
ClientProxyFactoryBean clientProxyFactoryBean = new ClientProxyFactoryBean();
clientProxyFactoryBean.setServiceClass(MyClass.class);
clientProxyFactoryBean.setWsdLocation(getAddress() + "?wsdl");
//OpenL databinding factory
AegisDatabindingFactoryBean aegisDatabindingFactoryBean = new AegisDatabindingFactoryBean();
//Set variations support. Recommend to use the same value as a project in server. Can't be
false, if service uses variations feature.
aegisDatabindingFactoryBean.setSupportVariations(true);
aegisDatabindingFactoryBean.setWriteXsiTypes(true);
//In case you need custom binding classes.
Set<String> overrideTypes = new HashSet<String>();
overrideTypes.add(<Some class>.class.getCanonicalName());
aegisDatabindingFactoryBean.setOverrideTypes(overrideTypes);
clientProxyFactoryBean.setDataBinding(aegisDatabindingFactoryBean.createAegisDatabinding());

MyClass myClass =(MyClass) clientProxyFactoryBean.create();
```

A dynamic client can be used for both static interface and dynamic interface generated in runtime configuration.

A dynamic client is a feature of CXF framework. For dynamic interface, use `JaxWsDynamicClientFactory` factory. For more information on using CXF for a dynamic interface, see CXF documentation.

The following example illustrates creation of a dynamic client:

```
JaxWsDynamicClientFactory dynamicClientFactory = JaxWsDynamicClientFactory.newInstance();
ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();
List<String> bindingFiles = new ArrayList<String>() {
    private static final long serialVersionUID = 1L;
    {
        add("binding.xml");
    }
};
Client client = dynamicClientFactory.createClient(<Service WSDL URL>, bindingFiles);
```

Binding.xml file content is as follows:

```
<jaxb:bindings version="2.2" xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
```

```
xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <jaxb:globalBindings generateElementProperty="false" collectionType="indexed"/>
</jaxb:bindings>
```

6.2 Using OpenL Tablets REST Services from Java Code

This section describes how to write a client code that invokes OpenL Tablets REST services projects. Another way can be used to invoke services but it is recommended to use Apache CXF framework to prevent additional effort for data binding.

The following example illustrates client code generation for the JSON content type:

```
JacksonObjectMapperFactoryBean jacksonObjectMapperFactoryBean = new
JacksonObjectMapperFactoryBean();
jacksonObjectMapperFactoryBean.setEnableDefaultTyping(true);
Set<String> overrideTypes = new HashSet<String>();
overrideTypes.add(SomeClass.class.getName());

jacksonObjectMapperFactoryBean.setOverrideTypes(overrideTypes);
ObjectMapper mapper = jacksonObjectMapperFactoryBean.createJacksonDataBinding();

final JacksonJsonProvider jsonProvider = new JacksonJsonProvider();

WebClient webClient = WebClient.create#REST service url#,
    new ArrayList<Object>() {
        private static final long serialVersionUID = 5636807402394548461L;
        {
            add(jsonProvider);
        }
    });

webClient.type(MediaType.APPLICATION_JSON);

Response response = webClient.get();
```

Note: If you use POST request for more than one argument, create a DTO that contains field with method argument names and send this DTO object via `webClient.post()` method.